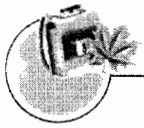

第4章

打印函数

目前，我们的操作系统已经完成了初始化的工作，现在可以正式进入系统运行阶段了。当然，在这之前我们必须首先解决一个基础性的问题——打印函数的问题。

一个功能强大的打印函数对于操作系统来说是至关重要的。操作系统作为底层软件，调试的时候往往要依赖于硬件调试工具，这本身就加大了操作系统调试的复杂度。对于那些高级操作系统来说，因为涉及到的问题非常复杂，即使有专业的调试工具，也很难实时捕获系统运行时的完整信息，因此，解决好调试的问题就成为了我们的操作系统编码过程中的首要问题。当然，最简单的往往最有效。一个强大的打印函数在大多数情况下足以承担操作系统的调试任务。所以，这里我们利用一章的篇幅来详细地分析一下打印函数的实现原理和方法，并在掌握了这些原理和方法之后亲自实现一个功能完整的打印函数。

在我们接触到的打印函数中，当属 printf 这个函数最为大家所熟悉，与之同系列的函数还包括 fprintf、sprintf、snprintf，等等。这一类函数通常被叫做格式化输出函数，是标准 C 库中不可或缺的一部分。这些函数的作用是都能够将“给定的内容”按照“指定格式”输出到“指定目标”内，比如 fprintf 和 sprintf 这两个函数都有类似的功能，而当指定的目标特指系统标准输出时就可以使用 printf 函数了。

明白了这一点，我们也就掌握了编写打印函数的关键点，那就是要处理

好如下三个问题。

- “给定的内容”
- “指定的格式”
- “指定目标”

以下面的这段代码为例：

```
int foo=1;
printf("%d",foo);
```

所谓“给定的内容”，指的是整型变量 foo，而“指定的格式”则是由 %d 来描述的，那么特定的目标呢？对于 printf 函数来说，标准输出就代表了特定目标。

处理好这三个问题，实现一个标准的打印函数便轻而易举了。然而，话虽如此，但如果现在就要求我们立刻给自己的操作系统实现一个标准输出函数，似乎还为时过早。我们不如先来学习一个打印函数的例子，掌握了基本原理和方法之后再动手不迟。

4.1 打印函数实例

代码 4-1 就是一个非常好的例子。

代码 4-1

```
void printf (const char *fmt, ...){
    va_list args;
    uint i;
    char printbuffer[CFG_PBSIZE];
    va_start (args, fmt);

    i = vsprintf (printbuffer, fmt, args);
    va_end (args);

    puts (printbuffer);
}
```

代码 4-1 选自 u-boot，u-boot 是高级嵌入式系统中比较常用的引导加载程序，可应用于多种硬件平台，支持许多嵌入式操作系统。尽管 u-boot 不是操作系统，不具有高级操作系统的基本结构，但是，作为一款优秀的引导程序，u-boot 具有功能强大、实现简单、结构清晰等诸多特点，非常适合学习和借鉴。于是，我们选用 u-boot 来进行标准输出函数的研究。

代码的第一行看起来就有些与众不同。printf 函数首先定义了一个只读的字符指针 const char *fmt，而剩下的参数则使用“...”来表示。我们都知道，这里的“...”表示 printf 函数的参数是可变的。该函数也正是使用了可变参数的处理方法才拥有了非常强大的功能。所以在学习 printf 函数之前，我们应该首先学习一下可变参数的原理和使用方法。

4.1.1　变参函数是如何工作的

我们知道，函数调用的参数传递需要依靠寄存器和堆栈来实现。C 语言默认的函数调用规范是所有参数从右到左依次入栈。这样，这些函数参数在弹出堆栈时，便能够获得一个从左到右的顺序。对于那些固定参数的函数，编译器清楚地知道要将多少个参数压入堆栈，在函数运行时依次从堆栈中弹出各个参数值然后使用。

然而，当传递的参数个数不确定时，编译器便不会知道参数的个数，也就不知道应该将多少个参数从堆栈中弹出。

因此，使用可变参数的函数，其参数不能仅包含可变参数，还至少需要一个参数来表示可变参数的个数。

举个例子来说，像下边这种函数形式是不允许的。此时，虽然函数参数可变，但当函数被调用时，程序却没有办法知道需要将多少个参数从堆栈中弹出才合适。因此，这样的代码是不能编译通过的。

```
void foo(...);
```

想要使用可变参数，其格式至少应该类似于如下格式。

```
void foo(int n,...);
```

这里，第一个参数的作用就是直接或间接地通知 foo 函数对 foo 函数的调用传递的参数有多少个，这样，程序至少在编译时不会报错。

若对函数 foo 的调用写成了如下形式：

```
foo(2,arg1,arg2);
```

这表示此次调用 foo 函数将传递两个参数，而写成下边的形式则代表传递 5 个参数：

```
foo(5,arg1,arg2,arg3,arg4,arg5);
```

当然，这里 foo 函数的第一个参数直接充当了传递参数个数的角色。很多时候，可变参数的个数并不是直接传递的，如 printf 函数。在代码 4-1 中，printf 函数的第一个参数是一个字符串，被打印的信息、打印格式以及可变参数个数都包含在这个字符串中。

定参函数和变参函数的调用过程如图 4-1 和图 4-2 所示。

了解完定参函数和变参函数的工作原理，接下来我们来学习一下如何在程序中使用变参函数。

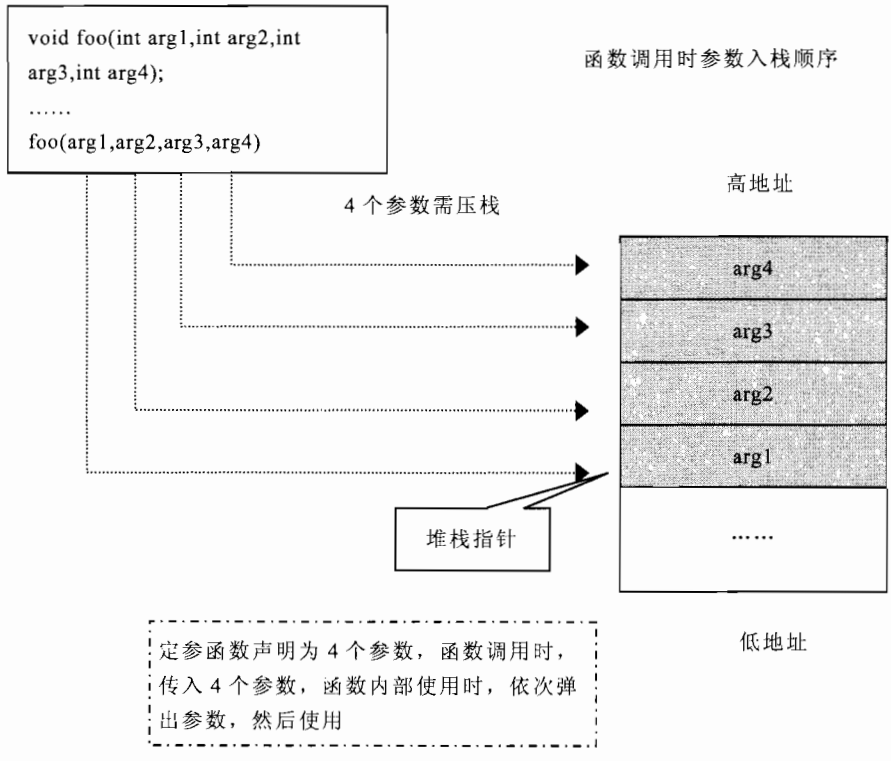

图 4-1　定参函数的调用过程

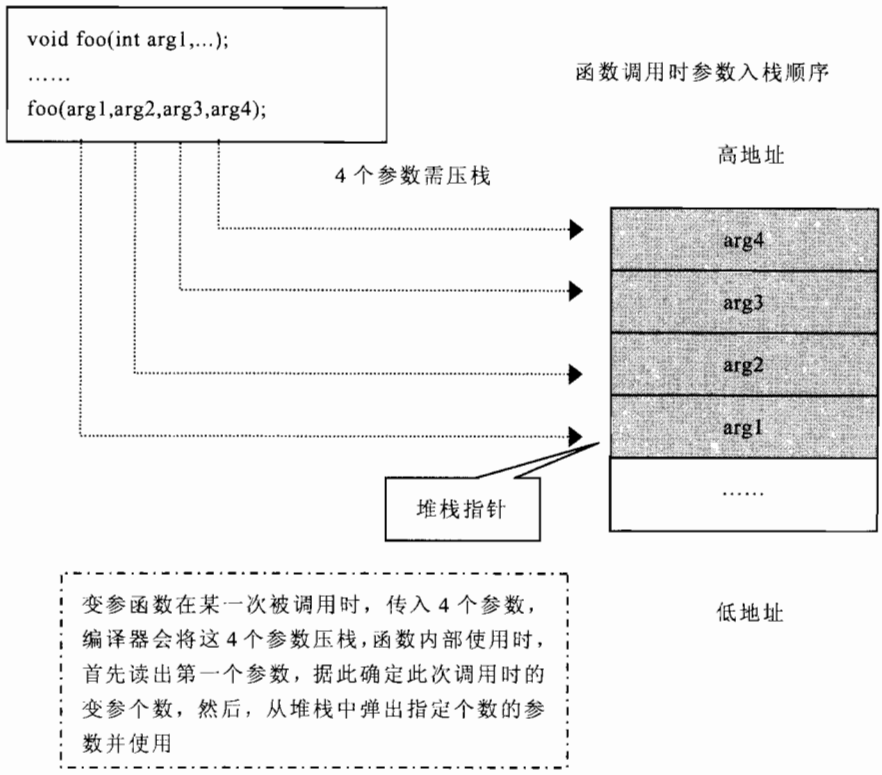

图 4-2　变参函数的调用过程

4.1.2　亲自实现一个可变参数函数

首先，关于变参函数声明方法，使用“...”来表示参数的可变部分，这在前面我们已经介绍过了。

接下来，就是使用一系列方法从变参函数里依次取出函数参数。在 C 语言中，我们有一组变量类型和宏专门用于处理变参函数，它们是 va_list、va_start、va_arg 和 va_end。对于这些变量类型和宏，典型定义如下：

代码 4-2

```
typedef char * va_list;
#define _INTSIZEOF(n) ((sizeof(n)+sizeof(int)-1)&~(sizeof(int)-1))
#define va_start(ap,v)    ( ap = (va_list)&v + _INTSIZEOF(v))
#define va_arg(ap,t)   (*(t*)((ap+=_INTSIZEOF(t)) - _INTSIZEOF(t)))
```

```
#define va_end(ap)      ( ap = (va_list)0 )
```

当然，代码 4-2 并不是唯一的定义方法。在不同的系统中，使用不同的编译器，包含有不同的标准库文件，其定义方法都各不相同。这里我们不急于分析每个宏的作用，首先通过一段实例来了解一下这些类型和宏的使用方法。

代码 4-3

```
void test_num(int num){
    *(char *)0xd0000020=num+'0';
}

void test_vparameter(int i,...){
    int c;
    va_list argv;
    va_start(argv,i);
    while(i--){
        c=va_arg(argv,int);
        test_num(c);
    }
    va_end(argv);
}
```

代码 4-3 是以我们已经完成的代码为基础的，结合第 3 章的程序，这段代码也可以正常运行。在程序中我们首先定义了一个变参函数 test_vparameter，该函数的第一个参数是一个整型值，表示在某次调用中传递给该函数的变参个数，而且要求所有变参的类型必须一致，这里我们人为地规定可变参数的类型都为 int。

在函数 test_vparameter 中，首先定义了一个 va_list 类型的变量 argv。

然后调用宏 va_start，这个宏定义的作用是得到可变参数列表中第一个参数的确切地址，并赋值给 argv。此时，指针变量 argv 就成功指向了函数 test_vparameter 参数堆栈的栈顶了。

接下来，函数在每次循环中读出一个参数，并将这些参数传递给 test_num 函数进行显示。test_num 函数只能简单地打印 0~9 中的数字作为测试函数，这已经足够了。循环控制依靠的是函数 test_vaparameter 中的第一个参数。当然，这里最关键的一条语句是 va_arg，大家可以从代码 4-2 中了解到它的定义。该宏以 va_list 变量作为第一个参数，以需要返回的变量的

类型作为第二个参数，这个宏的返回值就是可变参数列表中下一个待处理的参数。例如，在我们的例子里，已经规定了所有的可变参数的类型均为 int，所以在代码 4-3 中，其调用方法总是不变的，而每次宏 va_arg 返回时都将得到可变参数列表中的下一个参数，直至循环结束。

最后，函数 test_vparameter 调用了 va_end 宏结束对 va_list 变量的操作。

代码 4-3 中的例子描述了在 C 语言中操作变参函数的典型用法，由于去除了多余的代码，因此能够清晰地说明变参函数的基本结构。这里，我们把这种结构提取出来总结成表 4-1。

表 4-1　变参函数的使用结构

步骤	说明
va_list argv	定义一个变参变量 argv
va_start(argv,i)	初始化 argv
c=va_arg(argv,int)	在已知变量类型的情况下，获得下一个变参变量
va_end(argv)	结束变参变量操作

既然已经了解到表 4-1 中各个宏定义和变量类型的用法，那么我们不妨结合代码 4-2，深入研究一下这些宏定义究竟是怎样在变参函数中发挥作用的。

先来了解一下宏_INTSIZEOF(n)的作用，_INTSIZEOF(n)被定义成了如下形式：

```
((sizeof(n)+sizeof(int)-1)&~(sizeof(int) - 1) )
```

假设系统的 sizeof(int)的值为 4，那么经该宏计算后，会得到一个比 n 大的能整除 4 的数中的最小的那个数。例如，当 sizeof(n)＝1、2、3 或者 4 时，该宏的展开结果都为 4，而 sizeof(n)为 5、6、7 或 8 时，结果为 8。当然，这里 n 并不会随意取值，_INTSIZEOF(n)是用来计算各种数据类型按照 4 字节对齐后的结果的。也就是说，对于 char 和 short 类型的数据，因为不满一个 int 类型的内存空间，所以要按照 int 类型对齐，也就是 4 个字节，而对于类似于 long long 这种数据类型，则需要 8 个字节。

在宏_INTSIZEOF(n)的帮助下，我们可以根据一个变量的类型来计算该变量在内存中会占据多少个字节，从而正确定位内存中的参数位置。va_start(ap,v)就是一个非常典型的例子，它的定义如下：

```
( ap = (va_list)&v + _INTSIZEOF(v) )
```

va_start(ap,v)实际上就是在对变量 ap 进行赋值，其值为变量 v 的地址加上 v 的类型按照 4 字节对齐后的偏移。因为参数在内存中的分布是连续的，又因为传递给宏 va_start 的参数 v 都是变参函数里可变参数列表中的前一个参数。所以，变量 ap 最终的结果是一个指向变参列表首地址的指针。

紧接着在 va_start(ap,v)运行完成后，我们就可以使用宏 va_arg(ap,t)来依次提取变参列表中的参数，其定义如下：

```
( *(t *)((ap += _INTSIZEOF(t)) - _INTSIZEOF(t)) )
```

这里，宏 va_arg(ap,t)中的参数 t 只能输入变量类型，该宏恰恰也是根据变量的类型来正确提取参数的。首先，宏 va_arg 将变量 ap 的值在原有的基础上加上一个对齐后的偏移，此时，ap 已经指向变参到表中的下一个变量了，然后表达式在减去一个同样的值后被强行转成 t 类型，此时我们正好取出了变量 ap 所指向的那个参数，同时更新变量 ap，让它指向参数列表中的下一个参数以便下次使用。

最后，宏 va_end(ap)将变量 ap 赋值为零，从此不再使用。

这便是 C 语言中，变参函数的原理和基本操作方法。

本着在实践中学习的原则，接下来我们来运行一下代码 4-3。

首先，请在原有代码的基础上新建一个新的文件，命名为“print.c”。然后将代码 4-2 和代码 4-3 的内容复制到“print.c”文件中。接下来修改 Makefile 文件，将原来的：

```
OBJS=init.o start.o boot.o abnormal.o mmu.o
```

改为：

```
OBJS=init.o start.o boot.o abnormal.o mmu.o print.o
```

最后，将“boot.c”文件中的 plat_boot 函数修改为如下形式：

代码 4-4

```
void plat_boot(void){
extern void test_vparameter(int,...);
    int i;
    for(i=0;init[i];i++){
        init[i]();
    }
    init_sys_mmu();
```

```
    start_mmu();
    test_mmu();
    test_vparameter(3,9,8,7);
    test_vparameter(2,6,5);
    while(1);
}
```

编译源代码，并运行 skyeye 命令，在终端你将会看到如下结果：

```
......
arch: arm
cpu info: armv4, arm920t, 41009200, ff00fff0, 2
mach info: name s3c2410x, mach_init addr 0x806ad40
uart_mod:0, desc_in:, desc_out:, converter:
SKYEYE: use arm920t mmu ops
Loaded RAM   ./leeos.bin
start addr is set to 0x30000000 by exec file.
helloworld
test_mmu
98765
```

该结果正是我们两次调用 test_vparameter 函数的输出结果。

4.1.3　实现打印函数中的格式转换

在掌握了变参函数的用法之后，代码 4-1 的原理就不难理解了。在代码 4-1 中，printf 函数首先为变参函数的使用做了适当的准备，然后调用函数 vsprintf 将变参列表 args 中的变量按照 fmt 中规定的格式保存到临时缓存 printbuffer 中，最后再调用 puts(printfbuffer)函数将临时缓存中的字符串数据打印到终端中去。

在这个过程中，vsprintf 函数无疑起到了重要的作用，是整个 printf 函数的核心，下面就让我们来共同学习一下 vsprintf 函数的实现方法。

代码 4-5

```
int vsprintf(char *buf, const char *fmt, va_list args){
    unsigned NUM_TYPE num;
    int base;
    char *str;
    int flags;
    int field_width;
```

```
int precision;
int qualifier;
str = buf;

for (; *fmt ; ++fmt) {
    if (*fmt != '%') {
        *str++ = *fmt;
        continue;
    }

    /* process flags */
    flags = 0;
    repeat:
        ++fmt;      /* this also skips first '%' */
        switch (*fmt) {
            case '-': flags |= LEFT; goto repeat;
            case '+': flags |= PLUS; goto repeat;
            ……
        }
    ……

    base = 10;

    switch (*fmt) {
    case 'c':
    ……
        *str++ = (unsigned char) va_arg(args, int);
        ……
    continue;

    case 's':
        str = string(str, va_arg(args, char *),\
                        field_width, precision, flags);
        continue;
    ……
    case 'X':
        base = 16;
        break;

    case 'd':
    case 'i':
        flags |= SIGN;
    case 'u':
```

```
            break;

        default:
            *str++ = '%';
            if (*fmt)
                *str++ = *fmt;
            else
                --fmt;
            continue;
        }
        str = number(str, num, base, field_width, precision, flags);
    }
    *str = '\0';
    return str-buf;
}
```

代码 4-5 有些长，出于篇幅限制，我们删除了许多内容，只留下主干部分，以使结构更清晰。在讲解这段代码之前我们需要首先复习一下 printf 函数的格式符。

每一个懂 C 语言的人，都会了解表 4-2 的含义，每一本讲 C 语言的书都会涉及表 4-2 的内容。当然，printf 函数的输出格式符还远不止这些，这里我们也没有必要一一列举。

表 4-2　printf 函数格式符节选

常用格式符	说明
x	以 16 进制形式输出
u	以无符号整型形式输出
c	以字符形式输出
d	以有符号整型形式输出
s	以字符串的形式输出
o	以 8 进制形式输出

有的时候，打印函数并不需要支持所有的格式符，比如，在 Linux 内核当中，打印函数 printk 就不具备对浮点数的数据进行输出的能力。当然，表 4-2 中的这些格式符还可以进行组合变化。例如，在格式符前面加上“-”，表示字符串靠左，右侧补空格。如果在格式符前面出现“l”，表示以长类型输出，等等。

从学习的角度来讲，数据仍然是原来的数据，只不过我们需要根据不同的格式要求对同样的数据做不同的解析。而这种解析方法，针对数字、字符

或字符串，又完全不同。所以，我们只需要讲解几种格式的转换方法，就可以触类旁通。

vsprintf 函数的第一个参数指向了一段缓冲区，该缓冲区中的字符将会输出到终端上去。所以我们要做的就是，根据格式要求将需要转换的字符进行转换后放到这段缓冲区里，而将不需要转换的字符原样复制到缓冲区中。

我们知道，printf 函数中的格式符都需要以“%”开头，所以，代码 4-5 首先要将“%”（包括“%”之前的内容）剥离，得到待解析的字符。于是，程序需要在循环中依次读出缓存中的每一个字符，判断是否为“%”，如果是，就把它后边的字符作为解析格式，如果不是“%”，则将该字符原封不动地复制到缓冲区中，然后继续判断下一个字符。这些操作只需通过一个简单的 if 判断就能实现。

在读出“%”后边的字符之后，我们要判断该格式符是否是可变化的形式，比如前面是否带有“+”、“l”，等等。在代码 4-5 中，这是通过一个 switch…case 结构实现的。

在处理完所有的变化形式后，代码 4-5 就真正进入了格式解析的过程。该过程也是由一个 switch…case 结构实现的。printf 函数的格式符虽然很多，但基本上可以被分成三大类：数字、字符、字符串。其中，对字符的解析最简单，所以我们先来看一下字符解析。

解析字符之所以简单，是因为我们无须做转换。在代码 4-5 中，如果格式符为“c”，则将可变参数列表中的数据直接提取出来，强制转换成 char 类型后，直接放到缓冲区中。

如果待转换的数据类型是字符串，则从可变参数列表中拿到的将是指向该字符串的指针。那么我们需要做的工作就是，将该指针指向的数据复制到缓冲区中，所有这些工作都可以交给 string 这个函数来实现。

如果格式符是任何一种数字形式，那么我们就需要将可变参数列表中的数字根据不同的进制要求转换成字符串的形式，然后再进行复制。因此，在转换的过程中，我们只需要弄清楚两件事——进制和符号，就可以完成数字到字符串的转换了。在代码 4-5 里，进制数由一个名为 base 的变量保存，而符号则有 flag 标志保存。搞清楚这两个问题后，就可以调用 number 函数对数字做统一的处理了。

当可变参数列表中的所有参数都处理完毕后，vsprintf 函数也就完成了

它的任务，此时缓冲区中的数据已经转换完成，只需要将这些数据输出到终端即可。在代码 4-1 中，这最后的收尾工作是交给 puts 函数来完成的。对终端的定义，在不同的环境下是完全不同的，很多嵌入式系统并没有负责显示的设备，对于这样的系统，往往依靠串口来打印信息。u-boot 作为一个典型的嵌入式系统引导加载程序，串口也是作为默认终端出现的。因此，这里的 puts 函数其实就是将缓冲区中的数据顺序发送到串口 FIFO 寄存器中。

至此，u-boot 中的 printf 函数就介绍完了。

读者也可以读一下其他程序的打印函数，你会发现，不同环境下的打印函数从结构上看非常相似。这种现象并不是偶然的，打印函数无论在哪种环境里，其功能都是一致的，而像 u-boot 中的这种函数结构，又非常有利于功能的实现。所以，打印函数的这种结构也几乎成为了该类函数的一个标准实现方法。

4.2　实现自己的打印函数

在我们自己的操作系统里，也可以仿造 u-boot 中的打印函数来实现数据的格式化输出。这里，我们不妨根据打印函数的原理实现一个属于自己的版本，完整的代码如下。

代码 4-6

```
typedef char * va_list;
#define _INTSIZEOF(n)   ((sizeof(n)+sizeof(int)-1)&~(sizeof(int) - 1) )
#define va_start(ap,v)   ( ap = (va_list)&v + _INTSIZEOF(v) )
#define va_arg(ap,t)   ( *(t *)((ap += _INTSIZEOF(t)) - _INTSIZEOF(t)) )
#define va_end(ap)    ( ap = (va_list)0 )

const char *digits="0123456789abcdef";
char numbers[68];

static char print_buf[1024];

#define FORMAT_TYPE_MASK      0xff00
#define FORMAT_TYPE_SIGN_BIT  0x0100
```

```
#define FORMAT_TYPE_NONE      0x000
#define FORMAT_TYPE_CHAR      0x100
#define FORMAT_TYPE_UCHAR     0x200
#define FORMAT_TYPE_SHORT     0x300
#define FORMAT_TYPE_USHORT    0x400
#define FORMAT_TYPE_INT       0x500
#define FORMAT_TYPE_UINT      0x600
#define FORMAT_TYPE_LONG      0x700
#define FORMAT_TYPE_ULONG     0x800
#define FORMAT_TYPE_STR       0xd00
#define FORMAT_TYPE_PTR       0x900
#define FORMAT_TYPE_SIZE_T    0xb00

#define FORMAT_TYPE(x)    ((x)&FORMAT_TYPE_MASK)
#define SET_FORMAT_TYPE(x,t) do{\
 (x)&=~FORMAT_TYPE_MASK;(x)|=(t);\
}while(0)
#define FORMAT_SIGNED(x)  ((x)&FORMAT_TYPE_SIGN_BIT)

#define FORMAT_FLAG_MASK      0xffff0000
#define FORMAT_FLAG_SPACE     0x10000
#define FORMAT_FLAG_ZEROPAD   0x20000
#define FORMAT_FLAG_LEFT      0x40000
#define FORMAT_FLAG_WIDTH 0x100000

#define FORMAT_FLAG(x)           ((x)&FORMAT_FLAG_MASK)
#define SET_FORMAT_FLAG(x,f)  ((x)|=(f))

#define FORMAT_BASE_MASK      0xff
#define FORMAT_BASE_O         0x08
#define FORMAT_BASE_X         0x10
#define FORMAT_BASE_D         0x0a
#define FORMAT_BASE_B         0x02

#define FORMAT_BASE(x)    (FORMAT_BASE_MASK&(x))
#define SET_FORMAT_BASE(x,t)\
    do{(x)&=~FORMAT_BASE_MASK;(x)|=(t);}while(0)

#define do_div(n,base) ({ \
    int __res; \
    __res = ((unsigned int) n) % (unsigned int) base; \
    n = ((unsigned int) n) / (unsigned int) base; \
    __res; })
```

```
void __put_char(char *p,int num){
    while(*p&&num--){
        *(volatile unsigned int *)0xd0000020=*p++;
    };
}

void * memcpy(void * dest,const void *src,unsigned int count)
{
    char *tmp = (char *) dest, *s = (char *) src;
    while (count--)
        *tmp++ = *s++;
    return dest;
}

char *number(char *str, int num,int base,unsigned int flags){
    int i=0;
    int sign=0;

    if(FORMAT_SIGNED(flags)&&(signed int)num<0){
        sign=1;
        num=~num+1;
    }

    do{
        numbers[i++]=digits[do_div(num,base)];
    }while(num!=0);


    if(FORMAT_BASE(flags)==FORMAT_BASE_O){
        numbers[i++]='0';
    }else if(FORMAT_BASE(flags)==FORMAT_BASE_X){
        numbers[i++]='x';
        numbers[i++]='0';
    }else if(FORMAT_BASE(flags)==FORMAT_BASE_B){
        numbers[i++]='b';
        numbers[i++]='0';
    }
    if(sign)
        numbers[i++]='-';

    while (i-- > 0)
            *str++ = numbers[i];
```

```
    return str;
}

int format_decode(const char *fmt,unsigned int *flags){
    const char *start = fmt;

    *flags &= ~FORMAT_TYPE_MASK;
    *flags |= FORMAT_TYPE_NONE;
    for (; *fmt ; ++fmt) {
        if (*fmt == '%')
        break;
    }

    if (fmt != start || !*fmt)
        return fmt - start;

    do{
        fmt++;
        switch(*fmt){
            case 'l':
                SET_FORMAT_FLAG(*flags,FORMAT_FLAG_WIDTH);
                break;
            default:
                break;
        }
    }while(0);

    SET_FORMAT_BASE(*flags,FORMAT_BASE_D);
    switch (*fmt) {
        case 'c':
            SET_FORMAT_TYPE(*flags,FORMAT_TYPE_CHAR);
            break;

        case 's':
            SET_FORMAT_TYPE(*flags,FORMAT_TYPE_STR);
            break;

        case 'o':
            SET_FORMAT_BASE(*flags,FORMAT_BASE_O);
            SET_FORMAT_TYPE(*flags,FORMAT_TYPE_UINT);
            break;

        case 'x':
```

```
        case 'X':
            SET_FORMAT_BASE(*flags,FORMAT_BASE_X);
            SET_FORMAT_TYPE(*flags,FORMAT_TYPE_UINT);
            break;

        case 'd':
        case 'i':
            SET_FORMAT_TYPE(*flags,FORMAT_TYPE_INT);
            SET_FORMAT_BASE(*flags,FORMAT_BASE_D);
            break;
        case 'u':
            SET_FORMAT_TYPE(*flags,FORMAT_TYPE_UINT);
            SET_FORMAT_BASE(*flags,FORMAT_BASE_D);
            break;

    default:
        break;
    }
    return ++fmt-start;
}

int vsnprintf(char *buf, int size, const char *fmt, va_list args){
    int num;
    char *str, *end, c,*s;
    int read;
    unsigned int spec=0;

    str = buf;
    end = buf + size;

    if (end < buf) {
        end = ((void *)-1);
        size = end - buf;
    }

    while (*fmt) {
        const char *old_fmt = fmt;

        read = format_decode(fmt, &spec);
        fmt += read;

        if((FORMAT_TYPE(spec))==FORMAT_TYPE_NONE){
            int copy = read;
```

```
            if (str < end) {
                if (copy > end - str)
                    copy = end - str;
                    memcpy(str, old_fmt, copy);
            }
            str += read;

        }else if(spec&FORMAT_FLAG_WIDTH){
            //do nothing
        }else if(FORMAT_TYPE(spec)==FORMAT_TYPE_CHAR){
            c = (unsigned char) va_arg(args, int);
            if (str < end)
                *str = c;
            ++str;
        }else if(FORMAT_TYPE(spec)==FORMAT_TYPE_STR){
            s = (char *) va_arg(args, char *);
            while(str<end&&*s!='\0'){
                *str++=*s++;
            }
        }else{
            if(FORMAT_TYPE(spec)==FORMAT_TYPE_INT){
                num = va_arg(args, int);
            }else if(FORMAT_TYPE(spec)==FORMAT_TYPE_ULONG){
                num = va_arg(args, unsigned long);
            }else if(FORMAT_TYPE(spec)==FORMAT_TYPE_LONG){
                num = va_arg(args, long);
            }else if(FORMAT_TYPE(spec)==FORMAT_TYPE_SIZE_T){
                num = va_arg(args, int);
            }else if(FORMAT_TYPE(spec)==FORMAT_TYPE_USHORT){
                num = (unsigned short) va_arg(args, int);
            }else if(FORMAT_TYPE(spec)==FORMAT_TYPE_SHORT){
                num = (short) va_arg(args, int);
            }else{
                num = va_arg(args, unsigned int);
            }
            str=number(str,num,spec&FORMAT_BASE_MASK,spec);
        }
    }
    if (size > 0) {
        if (str < end)
            *str = '\0';
        else
```

```
            end[-1] = '\0';
        }
        return str-buf;
    }

    void printk(const char *fmt, ...)
    {
        va_list args;
        unsigned int i;

        va_start (args, fmt);
        i = vsnprintf (print_buf, sizeof(print_buf),fmt, args);
        va_end (args);

        __put_char (print_buf,i);
    }

    void test_printk(void){
        char *p="this is %s test";
        char c='H';
        int d=-256;
        int k=0;
        printk("testing printk\n");
        printk("test string ::: %s\ntest char ::: %c\ntest
digit ::: %d\ntest X ::: %x\ntest unsigned ::: %u\ntest zero :::
%d\n",p,c,d,d,d,k);
    }
```

代码 4-6 是一套相对完整的打印函数，我们按照操作系统源代码的一般原则，效仿 Linux 给它取名为 printk。这段打印函数功能相对较全，代码的组织与 uboot 中的 printf 函数稍有不同，但结构是一致的。因为前面我们已经对打印函数各个技术要点做了深入的讲解，所以这里就不对代码 4-6 进行具体分析了，相信读者读懂这段代码是没有问题的。

要运行这段代码，我们需要将其保存成文件，名字为“print.c”，用来替换原来的文件。

然后修改“boot.c”中的 plat_boot 函数，如下：

代码 4-7

```
void plat_boot(void){
    int i;
```

```
    for(i=0;init[i];i++){
        init[i]();
    }
    init_sys_mmu();
    start_mmu();
    test_mmu();
    test_printk();
    while(1);
}
```

为了验证打印函数的正确性，在代码 4-7 中我们使用了一个 test_printk 函数，将 printk 函数以各种形式进行数据输出。

在所有代码都整理完成后，在终端运行 make 命令进行编译，如果成功地生成了二进制文件，则运行 skyeye 命令，你将会看到如下结果：

```
……
arch: arm
cpu info: armv4, arm920t, 41009200, ff00fff0, 2
mach info: name s3c2410x, mach_init addr 0x806ad40
uart_mod:0, desc_in:, desc_out:, converter:
SKYEYE: use arm920t mmu ops
Loaded RAM   ./leeos.bin
start addr is set to 0x30000000 by exec file.
helloworld
test_mmu
testing printk
test string :::    this is %s test
test char ::: H
test digit ::: -256
test X ::: 0xffffff00
test unsigned ::: 4294967040
test zero ::: 0
```

至此，我们已经拥有了一个功能强大的打印函数，为我们的操作系统的后续实现进一步扫清了障碍。

4.3　总结

打个比方说，打印函数就像是计算机的一张嘴，从中可以将它的想法、情绪讲给我们听，这是计算机与人的一种最基本的交流手段。

相信读者都应该知道巴别塔的故事，在《圣经·旧约·创世记》第十一章中说，当时人类联合起来想要兴建希望能通往天堂的高塔，为了阻止人类的计划，上帝让人类说不同的语言，由于人类相互之间不能沟通，计划因此失败。从这个故事中我们也可以看出，交流在人与人之间是多么重要。

然而，人与计算机呢？毫无疑问，一个成功的交流方式会给人与计算机之间的关系带来巨大的变化。想一想计算机的发展历史吧，从最早的纸带输入到中期的文本模式的输出，再到现在的图形化窗口，精美的页面、炫酷的特效、语音、字符、图像等多种输入方式，人们在追求人与计算机之间的畅快交流的道路上是永无止境的。

由于硬件设施与本书篇幅的限制，我们不能够在我们的操作系统中实现人与机器间多元的沟通方式。然而，打印函数作为机器对人最原始最基本的交流手段，却是一个里程碑。从这一刻起，我们已经为属于我们自己的操作系统插上了翅膀，能否让它飞翔起来，就看您是否敢想敢做了！